reducing the incremental effort required for each house. If I know I am going to build a large number of very similar applications, they will have more commonalties that can be included in the framework rather than built individually.

Applications can override the framework-supplied functionality wherever appropriate. If a house framework came with pre-painted walls, the builder could just paint over them with preferred colors. Similarly, the object oriented principle of inheritance allows an application developer to override the behavior of the framework. In the paradigm of component-based reuse, key functionality is encapsulated in a component. The component can then be reused in multiple applications. In the house analogy, components correspond to appliances such as dishwashers, refrigerators, microwaves, etc. Similarly, many application components with pre-packaged functionality are available from a variety of vendors. An example of a popular component is a Data Grid It is a component that can be integrated into an application to deliver the capability of viewing columnar data in a spreadsheet-like grid. Component-based reuse is best suited for capturing *black-box-like* features, for example text processing, data manipulation, or any other features that do not require specialization.

Several applications on the same computer can share a single component. This is not such a good fit with the analogy, but imagine if all the houses in a neighborhood could share the same dishwasher simultaneously. Each home would have to supply its own dishes, detergent, and water, but they could all wash dishes in parallel. In the application component world, this type of sharing is easily accomplished and results in reduced disk and memory requirements.

Components tend to be less platform and tool dependent. A microwave can be used in virtually any house, whether it's framework is steel or wood, and regardless of whether it was customized for building mansions or shacks. You can put a high-end microwave in a low-end house and vice-versa. You can even have multiple different microwaves in your house Component technologies such as CORBA, COM, and Java Beans make this kind of flexibility commonplace in application development. Often, the best answer to achieving reuse is through a combination of framework-based and component-based techniques. A framework-based approach for building BusSim applications is appropriate for developing the user interface, handling user and system events, starting and stopping the application, and other application-specific and delivery platform-specific functions. A component-based approach is appropriate for black-box functionality. That is, functionality that can be used as-is with no specialization required. In creating architectures to support BusSim application development, it is imperative that any assets remain as flexible and extensible as possible or reusability may be diminished. Therefore, we chose to implement the unique aspects of BusSim applications using a component approach rather than a framework approach. This decision is further supported by the following observations.

Delivery Framework for Business Simulation

Components are combined with an Application Framework and an Application Architecture to achieve maximum reuse and minimum custom development effort. The Application Architecture is added to provide communication support between the application interface and the components, and between the components. This solution has the following features: The components (identified by the icons) encapsulate key BusSim functionality. The Application Architecture provides the glue that allows application-to-component and component-to-component communication. The Application Framework provides structure and base functionality that can be customized for different interaction styles. Only the application interface must be custom developed. The next section discusses each of these components in further detail.

The Business Simulation Toolset

We have clearly defined why a combined component/framework approach is the best solution for delivering high-quality BusSim solutions at a lower cost. Given that there are a number of third party frameworks already on the market that provide delivery capability for a wide variety of platforms, the TEL project is focused on defining and developing a set of

components that provide unique services for the development and delivery of BusSim solutions. These components along with a set of design and test workbenches are the tools used by instructional designers to support activities in the four phases of BusSim development. We call this suite of tools the Business Simulation Toolset. Following is a description of each of the components and workbenches of the toolset. A ***Component*** can be thought of as a black box that encapsulates the behavior and data necessary to support a related set of services. It exposes these services to the outside world through published interfaces. The published interface of a component allows you to understand what it does through the services it offers, but not how it does it. The complexity of its implementation is hidden from the user. The following are the key components of the BusSim Toolset. Domain Component - provides services for modeling the state of a simulation. Profiling Component - provides services for rule-based evaluating the state of a simulation. Transformation Component - provides services for manipulating the state of a simulation. Remediation Component - provides services for the rule-based delivering of feedback to the student The Domain Model component is the central component of the suite that facilitates communication of context data across the application and the other components. It is a modeling tool that can use industry-standard database such as Informix, Oracle, or Sybase to store its data. A ***domain model*** is a representation of the objects in a simulation. The objects are such pseudo tangible things as a lever the student can pull, a form or notepad the student fills out, a character the student interacts with in a simulated meeting, etc. They can also be abstract objects such as the ROI for a particular investment, the number of times the student asked a particular question, etc. These objects are called ***entities***. Some example entities include: Vehicles, operators and incidents in an insurance domain; Journal entries, cash flow statements and balance sheets in a financial accounting domain and Consumers and purchases in a marketing domain

An entity can also contain other entities. For example, a personal bank account entity might contain an entity that represents a savings account. Every entity has a set of ***properties*** where each property in some way describes the entity. The set of properties owned by an entity, in essence, define the entity. Some example properties include: An incident entity on an insurance application owns properties such as "Occurrence Date", "Incident Type Code", etc. A journal entry owns properties such as "Credit Account", "Debit Account", and "Amount"; and a revolving credit account entity on a mortgage application owns properties such as "Outstanding Balance", "Available Limit", etc. Figure 4 illustrates a small segment of a domain model for claims handlers in the auto insurance industry in accordance with a preferred embodiment.

Profiling Component

In the simplest terms, the purpose of the Profiling Component is to analyze the current state of a domain and identify specific things that are true about that domain. This information is then passed to the Remediation Component which provides feedback to the student. The Profiling Component analyzes the domain by asking questions about the domain's state, akin to an investigator asking questions about a case. The questions that the Profiler asks are called profiles. For example, suppose there is a task about building a campfire and the student has just thrown a match on a pile of wood, but the fire didn't start. In order to give useful feedback to the student, a tutor would need to know things like: was the match lit?, was the wood wet?, was there kindling in the pile?, etc. These questions would be among the profiles that the Profiling Component would use to analyze the domain. The results of the analysis would then be passed off to the Remediation Component which would use this information to provide specific feedback to the student. Specifically, a ***profile*** is a set of criteria that is matched against the domain. The purpose of a profile is to check whether the criteria defined by the profile is met in the domain. Using a visual editing tool, instructional designers create profiles to identify those things that are important to know about the domain for a given task. During execution of a BusSim application at the point that feedback is requested either by the student or pro-actively by the application, the set of profiles associated with the current task are evaluated to determine which ones are true. Example profiles

include: Good productions strategy but wrong Break-Even Formula; Good driving record and low claims history; and Correct Cash Flow Analysis but poor Return on Investment (ROI)

A profile is composed of two types of structures: characteristics and collective characteristics. A ***characteristic*** is a conditional (the *if* half of a rule) that identifies a subset of the domain that is important for determining what feedback to deliver to the student. Example characteristics include: Wrong debit account in transaction 1; Perfect cost classification; At Least 1 DUI in the last 3 years; More than $4000 in claims in the last 2 years; and More than two at-fault accidents in 5 years

A characteristic's conditional uses one or more atomics as the operands to identify the subset of the domain that defines the characteristic. An ***atomic*** only makes reference to a single property of a single entity in the domain; thus the term atomic. Example atomics include: The number of DUI's >= 1; ROI > 10%; and income between $75,000 and $110,000. A ***collective characteristic*** is a conditional that uses multiple characteristics and/or other collective characteristics as its operands. Collective characteristics allow instructional designers to build richer expressions (i.e., ask more complex questions). Example collective characteristics include: Bad Household driving record; Good Credit Rating; Marginal Credit Rating; Problems with Cash for Expense transactions; and Problems with Sources and uses of cash. Once created, designers are able to reuse these elements within multiple expressions, which significantly eases the burden of creating additional profiles. When building a profile from its elements, atomics can be used by multiple characteristics, characteristics can be used by multiple collective characteristics and profiles, and collective characteristics can be used by multiple collective characteristics and profiles. Figure **5** illustrates an insurance underwriting profile in accordance with a preferred embodiment.

Example Profile for Insurance Underwriting

Transformation Component - Whereas the Profiling Component asks questions about the domain, the Transformation Component performs calculations on the domain and feeds the results back into the domain for further analysis by the Profiling Component. This facilitates the modeling of complex business systems that would otherwise be very difficult to implement as part of the application. Within the Analysis phase of the Interface/Analysis/Interpretation execution flow, the Transformation Component actually acts on the domain before the Profiling Component does its analysis. The Transformation Component acts as a shell that wraps one or more data modeling components for the purpose of integrating these components into a BusSim application. The Transformation Component facilitates the transfer of specific data from the domain to the data modeling component (inputs) for calculations to be performed on the data, as well as the transfer of the results of the calculations from the data modeling component back to the domain (outputs). Figure **6** illustrates a transformation component in accordance with a preferred embodiment. The data modeling components could be third party modeling environments such as spreadsheet-based modeling (e.g., Excel, Formula1) or discrete time-based simulation modeling (e.g., PowerSim, VenSim). The components could also be custom built in C++, VB, Access, or any tool that is ODBC compliant to provide unique modeling environments. Using the Transformation Component to wrap a third party spreadsheet component provides an easy way of integrating into an application spreadsheet-based data analysis, created by such tools as Excel. The Transformation Component provides a shell for the spreadsheet so that it can look into the domain, pull out values needed as inputs, performs its calculations, and post outputs back to the domain.

For example, if the financial statements of a company are stored in the domain, the domain would hold the baseline data like how much cash the company has, what its assets and liabilities are, etc. The Transformation Component would be able to look at the data and calculate additional values like cash flow ratios, ROI or NPV of investments, or any other calculations to quantitatively analyze the financial health of the company. Depending on their complexity, these calculations could be performed by pre-existing spreadsheets that a client has already spent considerable time developing.